Part II

Results

CHAPTER 5

Sunrise

"They will speak with new tongues."

— Mark 16:17

"A wholesome tongue is a tree of life,

But perverseness in it breaks the spirit."

— Proverbs 15:4

In this chapter we describe the Sunrise programming language and its associated assertion language, which is the language studied in this work. This is a representative member of the family of imperative programming languages, and its constructs will be generally familiar to programmers. We have carefully chosen the constructs included to have natural, straightforward, and simple semantics, which will support proofs of correctness. To this end, we have extended the normal notation for some constructs, notably **while** loops and procedure declarations, to include annotations used in constructing the proof of a Sunrise program's correctness. These annotations are required, but have no effect on the actual operational semantics of the constructs involved. They could therefore be considered comments, except that they are used by the verification condition generator in order to produce an appropriate set of verification conditions to complete the proof.

In the past, there has been considerable debate over the need for the programmer to provide, say, a loop invariant. Some have claimed that this is an unreasonable burden on the programmer, who should have to provide only a program and an input/output specification. Others have replied that the requirement to provide a loop invariant forces clear thinking and documentation that should have been done in any case.

We would like to take the pragmatic position that the provision of loop invariants is necessary for the simple definition of verification condition generators, which are not complex functions. The same principle holds for the more complex annotations we require for procedures, that the provision of these annotations are necessary for simple and clean definitions of the program logic rules which serve as an axiomatic semantics for procedures. If one wishes to transfer the burden of coming up with the loop invariant from the human to the automatic computer, one incurs a great increase in the degree of difficulty of constructing the verification condition generator, including the need for automatic theorem provers, multiple decision procedures, and search strategies which have exponential time complexity. We wish to attempt something rather more tractable, and to perform only part of the task, in particular that part which seems most amenable to automatic analysis. This desire has guided the construction of the language here defined.

`exp:`	e	$::=$	$n \mid x \mid {+}{+}x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$
`(exp)list:`	es	$::=$	$\langle\,\rangle \mid CONS\ e\ es$
`bexp:`	b	$::=$	$e_1 = e_2 \mid e_1 < e_2 \mid es_1 \ll es_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b$
`cmd:`	c	$::=$	**skip** $\mid$ **abort** $\mid$ $x := e$ $\mid$ $c_1\ ;\ c_2$ $\mid$ **if** b **then** c_1 **else** c_2 **fi** $\mid$ **assert** a **with** a_{pr} **while** b **do** c **od** $\mid$ $p(x_1,\ \ldots,\ x_n\ ;\ e_1,\ \ldots,\ e_m)$

`decl:`	d	$::=$	**procedure** p (**var** $x_1,\ \ldots,\ x_n$; **val** $y_1,\ \ldots,\ y_m$);
			    global $z_1,\ \ldots,\ z_k$;
			    pre a_{pre};
			    post a_{post};
			    calls p_1**with** a_1;
			        $\vdots$
			    calls p_j**with** a_j;
			    recurses with a_{rec};
			    c
			end procedure
			$\mid$ $d_1\ ;\ d_2$
			$\mid$ **empty**

`prog:`	π	$::=$	**program** $d\,; c$ **end program**

Table 5.1: Sunrise programming language.

5.1 Programming Language Syntax

Table 5.1 contains the concrete syntax of the Sunrise programming language, defined using Backus-Naur Form as a context-free grammar.

We define six types of phrases in this programming language (Table 5.2):

Type	Description	Typical Member
`exp`	numeric expressions	e
`(exp)list`	lists of numeric expressions	es
`bexp`	boolean expressions	b
`cmd`	commands	c
`decl`	declarations	d
`prog`	programs	π

Table 5.2: Sunrise programming language types of phrases.

The lexical elements of the syntax expressed in Table 5.1 are numbers and variables. Numbers (denoted by n) are simple unsigned decimal integers, including zero, with no *a priori* limit on size, to match the HOL type `num`. They cannot be negative, either as written or as the result of calculations.

Variables (denoted with x or y, etc.) are a new concrete datatype `var` consisting of two components, a string and a number. In HOL a character string may be of any length from zero or more. The name of a variable is typically printed as the string, followed immediately by the variant number, unless it is zero, when no number is printed; the possibility exists for ambiguity of the result. The parser we have constructed expects the name of the variable to consist of letters, digits, and underscore ('_'), except that the first character may also be a caret ('^'). However, the operations of the VCG allow the string to contain any characters. The meaning of the string is to be the base of the name of the variable, and the meaning of the number is to be the variant number of the variable. Hence there might be several variables with the same string but differing in their number attribute, and these are considered distinct variables. This structure is used for variables to ease the construction of variants of variables, by simply changing (increasing) the variant number of the variable.

Variables are divided into two classes, depending on the initial character (if any) of the string. If the initial character is a caret ('`^`'), then the variable is a *logical variable*, otherwise it is a *program variable*. Program and logical variables are completely disjoint; "`y`" and "`^y`" are separate and distinct variables. Both kinds are permitted in assertion language expressions, but only program variables are permitted in programming language expressions. Since logical variables cannot appear in programming language expressions, they may never be altered by program control, and thus retain their values unchanged throughout a computation.

The syntax given in Table 5.1 uses standard notations for readability. The actual data types (except for lists) are created in HOL as new concrete recursive datatypes, using Melham's type definition package [GM93]. The results of this definition includes the creation of the constructor functions for the various programming language syntactic phrases in Table 5.3. This forms the abstract syntax of the Sunrise programming language.

All the internal computation of the verification condition generator is based on manipulating expressions which are trees of these constructor functions and the corresponding ones for assertion language expressions. These trees are not highly legible. However, we have provided parsers and pretty-printing functions

`exp` :	$NUM\ n$	n
	$PVAR\ x$	x
	$INC\ x$	$++x$
	$PLUS\ e_1\ e_2$	$e_1 + e_2$
	$MINUS\ e_1\ e_2$	$e_1 - e_2$
	$MULT\ e_1\ e_2$	$e_1 * e_2$
`bexp` :	$EQ\ e_1\ e_2$	$e_1 = e_2$
	$LESS\ e_1\ e_2$	$e_1 < e_2$
	$LLESS\ es_1\ es_2$	$es_1 \ll es_2$
	$AND\ b_1\ b_2$	$b_1 \wedge b_2$
	$OR\ b_1\ b_2$	$b_1 \vee b_2$
	$NOT\ b$	$\sim b$
`cmd` :	$SKIP$	**skip**
	$ABORT$	**abort**
	$ASSIGN\ x\ e$	$x := e$
	$SEQ\ c_1\ c_2$	$c_1\ ;\ c_2$
	$IF\ b\ c_1\ c_2$	**if** b **then** c_1 **else** c_2 **fi**
	$WHILE\ a\ pr\ b\ c$	**assert** a **with** pr **while** b **do** c **od**
	$CALL\ p\ xs\ es$	$p(xs; es)$
`decl` :	$PROC\ p\ vars\ vals\ glbs$ $pre\ post\ calls\ rec\ c$	**proc** $p\ vars\ vals\ glbs$ $pre\ post\ calls\ rec\ c$
	$DSEQ\ d_1\ d_2$	$d_1\ ;\ d_2$
	$DEMPTY$	**empty**
`prog` :	$PROG\ d\ c$	**program** $d; c$ **end program**

Table 5.3: Sunrise programming language constructor functions.

to provide an interface that is more human-readable, so that the constructor trees are not seen for most of the time.

5.2 Informal Semantics of Programming Language

The constructs in the Sunrise programming language, shown in Table 5.1, are mostly standard. The full semantics of the Sunrise language will be given as a structural operational semantics later in this chapter. But to familiarize the reader with these constructs in a more natural and understandable way, we here give informal descriptions of the semantics of the Sunrise language. This is intended to give the reader the gist of the meaning of each operator and clause in Table 5.1. We also describe the significance of the system of annotations for both partial and total correctness.

5.2.1 Numeric Expressions

n is an unsigned integer.

x is a program variable. It may not here be a logical variable.

$++x$ denotes the increment operation, where x is a program variable as above. The increment operation adds one to the variable, stores that new value into the variable, and yields the new value as the result of the expression.

The addition, subtraction, and multiplication operators have their normal meanings, except that subtraction is restricted to nonnegative values, so $x - y = 0$ for $x \leq y$. The two operands of a binary operator are evaluated in order from left to right, and then their values are combined and the numeric result yielded.

5.2.2 Lists of Numeric Expressions

HOL provides a polymorphic list type, and a set of list operators that function on lists of any type. This list type has two constructors, NIL and $CONS$, with the standard meanings. In both its meta language and object language, HOL typically displays lists using a more compact notation, using square brackets ([ ]) to delimit lists and semicolons (;) to separate list elements. Thus $NIL = [\,]$, and [2;3;5;7] is the list of the first four primes. In this programming language we wish to reserve square brackets to denote total correctness specifications, and so we will use angle brackets ($\langle\,\rangle$) instead to denote lists within the Sunrise language, for example $\langle 2; 3; 5; 7\rangle$ or $\langle\,\rangle$. When dealing with HOL lists, however, the square brackets will still be used.

The numeric expressions in a list are evaluated in order from left to right, and their values are combined into a list of numbers which is the result yielded.

5.2.3 Boolean Expressions

The operators provided here have their standard meaning, except for $es_1 \ll es_2$, which evaluates two lists of expressions and compares their values according to their lexicographic ordering. Here the left-most elements of each list are compared first, and if the element from es_1 is less, then the test is true; if the element from es_1 is greater, then the test is false; and if the element from es_1 is the same as (equal to) the element from es_2, then these elements are discarded and the tails of es_1 and es_2 are compared in the same way, recursively.

For every operator here, the operands are evaluated in order from left to right, and their values combined and the boolean result yielded.

5.2.4 Commands

The **skip** command has no effect on the state. **abort** causes an immediate abnormal termination of the program. $x := e$ evaluates the numeric expression e and assigns the value to the variable x, which must be a program variable. $c_1 \; ; \; c_2$ executes command c_1 first, and if it terminates, then executes c_2. The conditional command **if** b **then** c_1 **else** c_2 **fi** first evaluates the boolean expression b; if it is true, then c_1 is executed, otherwise c_2 is executed.

The iteration command **assert** a **with** a_{pr} **while** b **do** c **od** evaluates b; if it is true, then the body c is executed, followed by executing the whole iteration command again, until b evaluates to false, when the loop ends. The "**assert** a" and "**with** a_{pr}" phrases of the iteration command do not affect its execution; these are here as annotations to aid the verification condition generator. a denotes an *invariant*, a condition that is true every time control passes through the head of the loop. This is used in proving the partial correctness of the loop.

In contrast, a_{pr} denotes a *progress expression*, which here must be of the form $v < x$, where v is a assertion language numeric expression and x is a logical variable. v may only contain program variables. Assertion language expressions will be defined presently; here, v serves as a *variant*, an expression whose value strictly decreases every time control passes through the head of the loop. This is used in proving the termination of the loop. In future versions of the Sunrise programming language, we intend to broaden a_{pr} to other expressions, such as $vs \ll xs$, whose variants describe values of well-founded sets.

Finally, $p(x_1, \ldots, x_n \; ; \; e_1, \ldots, e_m)$ denotes a procedure call. This first evaluates the actual value parameters $e_1, \ldots, e_m$ in order from left to right, and then calls procedure p with the resulting values and the actual variable parameters $x_1, \ldots, x_n$. The value parameters are passed by value; the variable parameters are passed by name, to simulate call-by-reference. The call must match the declaration of p in the number of both variable and value parameters. Aliasing is forbidden, that is, the actual variable parameters $x_1, \ldots, x_n$ may not contain any duplicates, and may not duplicate any global variables accessible from p. The body of p has the actual variable parameters substituted for the formal variable parameters. This substituted body is then executed on the state where the values from the actual value parameters have been bound to the formal value parameters. If the body terminates, then at the end the values of the formal value parameters are restored to their values before the procedure was entered. The effect of the procedure call is felt in the actual variable parameters and in the globals affected.

5.2.5 Declarations

The main kind of declaration is the procedure declaration; the other forms simply serve to create lists of procedure declarations or empty declarations. The procedure declaration has the concrete syntax

```
procedure $p$ (var $x_1, \ldots, x_n$ ; val $y_1, \ldots, y_m$);
    global $z_1, \ldots, z_k$;
    pre $a_{pre}$;
    post $a_{post}$;
    calls $p_1$with $a_1$;
          ⋮
    calls $p_j$with $a_j$;
    recurses with $a_{rec}$;
    $c$
end procedure
```

This syntax is somewhat large and cumbersome to repeat; we will usually use instead the lithe abstract syntax version

$$\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c$$

where it is to be understood that we mean

$$\begin{array}{rcl} p & = & p \\ vars & = & x_1,\ \ldots,\ x_n \\ vals & = & y_1,\ \ldots,\ y_m \\ vars & = & z_1,\ \ldots,\ z_k \\ pre & = & a_{pre} \\ post & = & a_{post} \\ calls & = & (\lambda p.\ \mathbf{false})[a_j/p_j]\ldots[a_1/p_1] \\ rec & = & a_{rec} \\ c & = & c \end{array}$$

Note that the *calls* parameter is now a *progress environment* of type `prog_env`, where `prog_env` = `string` → `aexp`, a function from procedure names to progress expressions, to serve as the collection of all the **calls** ... **with** phrases given.

The meaning of each one of these parameters is as follows:

- p is the name of the procedure, a simple string.
- $vars$ is the list of the formal variable parameters, a list of variables. If there are no formal variable parameters, the entire "**var** $x_1, \ldots, x_n$" phrase may be omitted.
- $vals$ is the list of the formal value parameters, a list of variables. If there are no formal value parameters, the entire "**val** $y_1, \ldots, y_m$" phrase may be omitted.
- $glbs$ is the list of the global variables accessible from this procedure. This includes not only those variables read or written within the body of this procedure, but also those read or written by any procedure called immediately or eventually by the body of this procedure. Thus it is a list of all globals which can possibly be read or written during the course of execution of the body once entered. If there are no globals accessible, the entire "**global** $z_1, \ldots, z_k$;" phrase may be omitted.
- pre is the precondition of this procedure. This is a boolean expression in the assertion language, which denotes a requirement that must be true whenever the procedure is entered. Only program variables may be used.
- $post$ is the postcondition of this procedure. This is a boolean expression in the assertion language, which denotes the relationship between the states at the entrance and exit of this procedure. Two kinds of variables may be used in $post$, program variables and logical variables. The logical variables

will denote the values of variables at the time of entrance, and the program variables will denote the values of the variables at the time of exit. The postcondition expresses the logical relationship between these two sets of values, and thus describes the effect of calling the procedure.

- *calls* is the progress environment, the collection of all the **calls** ... **with** phrases given. Each "**calls** p_i **with** a_i" phrase expresses a relationship between two states, similar to the *post* expression but for different states. The first state is that at the time of entrance of this procedure p. The second state is that at any time that procedure p_i is called directly from the body of p. That is, if while executing the body of p there is a direct call to p_i, then the second state is that just after entering p_i.

  Expression a_i is a *progress expression.* Similar to the *post* expression, there are two kinds of variables that may be used in a_i, program variables and logical variables. The logical variables will denote the values of variables at the time of entrance of p, and the program variables will denote the values of the variables at the time of entrance of p_i. The progress expression gives the logical relationship between these two sets of values, and thus describes the degree of progress achieved between these calls.

- *rec* is the *recursion expression* for this procedure. It is a progress expression, similar to the progress expression of an iteration command, describing a relationship between two states. For *rec*, the first state is that at the time of entrance of p, and the second state is any time of entrance of p recursively as part of executing the body of p for the first call.

  Similar to the *post* expression, there are two kinds of variables that may be used in rec, program variables and logical variables. The logical variables will denote the values of variables at the time of original entrance of p, and the program variables will denote the values of the variables at the times of recursive entrance of p. The rec expression gives the logical relationship between these two sets of values, and thus describes the degree of progress achieved between recursive calls.

There are two permitted forms for rec. rec may be of the form $v < x$, where v is an assertion language numeric expression and x is a logical variable, or rec may be **false**. **false** is appropriate when the procedure p is not recursive and cannot call itself. Otherwise, $v < x$ should be used. v may only contain program variables; it serves as a variant, an expression whose value strictly decreases for each recursive call. Thus if v was equal to x at the time rec was originally called, then at any recursive call to p nested within that first call to p, we should have $v < x$.

In the future we intend to broaden this to include other expressions, such as $vs \ll xs$, whose variants describe values in well-founded sets, and the strict decrease described will be in terms of the relation used, e.g., $\ll$.

If this procedure is not expected to ever call itself recursively, then the phrase "**recurses with** a_{rec};" may be omitted, in which case rec is taken by default to be **false**.

- Command c is the body of this procedure. It may only use variables appearing in $vars$, $vals$, or $glbs$.

The actual significance of the various annotations, especially $calls$ and rec, will be explained in greater depth and illustrated with examples in later chapters.

5.2.6 Programs

A program consists of a declaration of a set of procedures and a command as the main body. The declarations are processed to create a *procedure environment* ρ of type `env`, collecting all of the information declared for each procedure into a function from procedure names to tuples of the following form:

$$\rho\ p \ = \ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle.$$

The definition of `env` is

$$\begin{aligned}\texttt{env} \ = \ \texttt{string} \ \rightarrow \ & ((\texttt{var})\texttt{list} \times (\texttt{var})\texttt{list} \times (\texttt{var})\texttt{list} \times \\ & \ \texttt{aexp} \times \texttt{aexp} \times \texttt{prog\_env} \times \texttt{aexp} \times \texttt{cmd}).\end{aligned}$$

This environment is the context used for executing the bodies of the procedures themselves, and also for executing the main body of the program.

The program is considered to begin execution in a state where the value of all variables is zero; however, this initial state is not included in the proof of a program's correctness. A future version of the Sunrise program may have an arbitrary initial state, and the same programs will prove correct.

5.3 Assertion Language Syntax

Table 5.4 contains the syntax of the Sunrise assertion language, defined using Backus-Naur Form as a context-free grammar.

We define three types of phrases in this assertion language, in Table 5.5.

The above syntax uses standard notations for readability. The actual data types are created in HOL as new concrete recursive datatypes, using Melham's

`vexp:`	v	$::=$	$n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$
`(vexp)list:`	vs	$::=$	$\langle\,\rangle \mid CONS\ v\ vs$
`aexp:`	a	$::=$	$\mathbf{true} \mid \mathbf{false}$
		$\mid$	$v_1 = v_2 \mid v_1 < v_2 \mid vs_1 \ll vs_2$
		$\mid$	$a_1 \wedge a_2 \mid a_1 \vee a_2 \mid \sim a$
		$\mid$	$a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid (a_1 => a_2 \mid a_3)$
		$\mid$	$\mathbf{close}\ a \mid \forall x.\ a \mid \exists x.\ a$

Table 5.4: Sunrise assertion language.

Type	Description	Typical Member
`vexp`	numeric expressions	v
`(vexp)list`	lists of numeric expressions	vs
`aexp`	boolean expressions	a

Table 5.5: Sunrise assertion language types of phrases.

type definition package [GM93]. The results of this definition includes the creation of the constructor functions for the various assertion language syntactic phrases in Table 5.6. This forms the abstract syntax of the Sunrise assertion language.

`vexp` :	$ANUM\ n$	n
	$AVAR\ x$	x
	$APLUS\ v_1\ v_2$	$v_1 + v_2$
	$AMINUS\ v_1\ v_2$	$v_1 - v_2$
	$AMULT\ v_1\ v_2$	$v_1 * v_2$
`aexp` :	$ATRUE$	**true**
	$AFALSE$	**false**
	$AEQ\ v_1\ v_2$	$v_1 = v_2$
	$ALESS\ v_1\ v_2$	$v_1 < v_2$
	$ALLESS\ vs_1\ vs_2$	$vs_1 \ll vs_2$
	$AAND\ a_1\ a_2$	$a_1 \wedge a_2$
	$AOR\ a_1\ a_2$	$a_1 \vee a_2$
	$ANOT\ a$	$\sim a$
	$AIMP\ a_1\ a_2$	$a_1 \Rightarrow a_2$
	$AEQB\ a_1\ a_2$	$a_1 = a_2$
	$ACOND\ a_1\ a_2\ a_3$	$a_1 => a_2 \mid a_3$
	$ACLOSE\ a$	**close** a
	$AFORALL\ x\ a$	$\forall x.\ a$
	$AEXISTS\ x\ a$	$\exists x.\ a$

Table 5.6: Sunrise assertion language constructor functions.

5.4 Informal Semantics of Assertion Language

The constructs in the Sunrise assertion language, shown in Table 5.4, are mostly standard. The full semantics of the Sunrise assertion language will be given as a denotational semantics later in this chapter. But to familiarize the reader with these constructs in a more natural and understandable way, we here give informal descriptions of the semantics of the Sunrise assertion language. This is intended to give the reader the gist of the meaning of each operator and clause.

The evaluation of any expression in the assertion language cannot change the state; hence it is immaterial in what order subexpressions are evaluated.

5.4.1 Numeric Expressions

n is an unsigned integer, as before for the programming language.

x is a variable, which may be either a program variable or a logical variable.

The addition, subtraction, and multiplication operators have their normal meanings, except that subtraction is restricted to nonnegative values, so $x - y = 0$ for $x \leq y$.

5.4.2 Lists of Numeric Expressions

These are similar to the lists of numeric expressions described previously for the programming language, except that the constituent expressions are assertion language numeric expressions. This list type has two constructors, NIL and $CONS$, with the standard meanings.

5.4.3 Boolean Expressions

Most of the operators provided here have their standard meaning, and are similar to their counterparts in the programming language, if one exists. **true** and

false are the logical constants. $=$ and $<$ have the normal interpretation, and so do the various boolean operators, such as conjunction ($\wedge$) and disjunction ($\vee$). $vs_1 \ll vs_2$ evaluates two lists of expressions and compares their values according to their lexicographic ordering. $(a_1 \Rightarrow a_2 \mid a_3)$ is a conditional expression, first evaluating a_1, and then yielding the value of a_2 or a_3 respectively, depending on whether a_1 evaluated to T or F, which are the HOL truth constants. **close** a forms the universal closure of a, which is true when a is true for all possible assignments to its free variables. We have specifically included the universal and existential quantifiers; all quantification is over the nonnegative integers.

5.5 Formal Semantics

"There are, it may be, so many kinds of languages in the world, and none of them is without significance. Therefore, if I do not know the meaning of the language, I shall be a foreigner to him who speaks, and he who speaks will be a foreigner to me."
— 1 Corinthians 14:10, 11

We present in this section the structural operational semantics of the Sunrise programming language, according to the style of Plotkin [Plo81] and Hennessey [Hen90]. We also present the semantics of the Sunrise assertion language in a denotational style.

The definitions in this section are the primary foundation for all succeeding proof activity. In particular, it is from these definitions that the five program logics described in Chapter 6 are proven sound, and from which the verification condition generator presented in Chapter 7 is proven sound. It is therefore also the foundation for the example programs which are verified in Chapter 8.

These extensions to the HOL system are purely definitional. No new axioms are asserted. This is therefore classified as a "conservative extension" of HOL, and there is no possibility of unsoundness entering the system. This security was essential to our work. This choice ensured that we faced a very difficult task in proving the soundness of the logics of Chapter 6, and in fact this may have consumed 65–70% of the effort of this project. These proofs culminated in the VCG soundness theorems, and once proven, the theorems are applied to example programs without needing to retrace the same proofs for each example.

This significant expenditure of effort was necessary because of the history of unsoundness in proposed axiomatic semantics, particularly in relation to procedures. After constructing the necessary proofs, we are grateful for the unrelenting vigilance of the HOL system, which kept us from proving any incorrect theorems. Apparently it is easier to formulate a correct structural operational semantics than it is to formulate a sound axiomatic semantics. This agrees with our intuition, that an axiomatic semantics is inherently higher-level than operational semantics, and omits details covered at the lower level. We exhibit this structural operational semantics as the critical foundation for our work, and present it for the research community's appraisal.

As previously described, the programming language has six kinds of phrases, and the assertion language has three. For each programming language phrase, we define a relation to denote the semantics of that phrase. The structural operational semantics consists of a series of rules which together constitute an inductive definition of the relation. This is implemented in HOL using Melham's excellent library [Mel91] for inductive rule definitions.

The semantics of the assertion language is defined in a denotational style. For each assertion language phrase, we define a function which yields the interpretation of that phrase into the HOL Object Language. This is implemented in HOL using Melham's tool for defining recursive functions on concrete recursive types [Mel89]. The types used here are the types of the assertion language phrases.

5.5.1 Programming Language Structural Operational Semantics

The structural operational semantics of the six kinds of Sunrise programming language phrases is expressed by the six relations in Table 5.7.

$E\ e\ s_1\ n\ s_2$	numeric expression e:`exp` evaluated in state s_1 yields numeric value n:`num` and state s_2
$ES\ es\ s_1\ ns\ s_2$	numeric expressions es:`(exp)list` evaluated in state s_1 yield numeric values ns:`(num)list` and state s_2
$B\ b\ s_1\ t\ s_2$	boolean expression b:`bexp` evaluated in state s_1 yields truth value t:`bool` and state s_2
$C\ c\ \rho\ s_1\ s_2$	command c:`cmd` evaluated in environment ρ and state s_1 yields state s_2
$D\ d\ \rho_1\ \rho_2$	declaration d:`decl` elaborated in environment ρ_1 yields result environment ρ_2
$P\ \pi\ s$	program π:`prog` executed yields state s

Table 5.7: Sunrise programming language semantic relations.

In Table 5.8, we present rules that inductively define the numeric expression semantic relation E. This is a structural operational semantics for numeric expressions.

Number:

$$\frac{}{E\ (n)\ s\ n\ s}$$

Variable:

$$\frac{}{E\ (x)\ s\ (s\ x)\ s}$$

Increment:

$$\frac{E\ (x)\ s_1\ n\ s_2}{E\ (\texttt{++}x)\ s_1\ (n+1)\ s_2[(n+1)/x]}$$

Addition:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1+e_2)\ s_1\ (n_1+n_2)\ s_3}$$

Subtraction:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1-e_2)\ s_1\ (n_1-n_2)\ s_3}$$

Multiplication:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1*e_2)\ s_1\ (n_1*n_2)\ s_3}$$

Table 5.8: Numeric Expression Structural Operational Semantics.

In Table 5.9, we present rules that inductively define the numeric expression list semantic relation ES. This is a structural operational semantics for lists of numeric expressions. The ES relation was actually defined in HOL as a list

Nil:

$$\frac{}{ES\ (\langle\rangle)\ s\ []\ s}$$

Cons:

$$\frac{E\ e\ s_1\ n\ s_2 \qquad ES\ es\ s_2\ ns\ s_3}{ES\ (CONS\ e\ es)\ s_1\ (CONS\ n\ ns)\ s_3}$$

Table 5.9: Numeric Expression List Structural Operational Semantics.

recursive function, with two cases for the definition based on NIL or $CONS$.

In Table 5.10, we present rules that inductively define the boolean expression semantic relation B. This is a structural operational semantics for boolean expressions.

Equality:

$$\frac{\begin{array}{c} E\ e_1\ s_1\ n_1\ s_2 \\ E\ e_2\ s_2\ n_2\ s_3 \end{array}}{B\ (e_1 = e_2)\ s_1\ (n_1 = n_2)\ s_3}$$

Less Than:

$$\frac{\begin{array}{c} E\ e_1\ s_1\ n_1\ s_2 \\ E\ e_2\ s_2\ n_2\ s_3 \end{array}}{B\ (e_1 < e_2)\ s_1\ (n_1 < n_2)\ s_3}$$

Lexicographic Less Than:

$$\frac{\begin{array}{c} ES\ es_1\ s_1\ ns_1\ s_2 \\ ES\ es_2\ s_2\ ns_2\ s_3 \end{array}}{B\ (es_1 \ll es_2)\ s_1\ (ns_1 \ll ns_2)\ s_3}$$

Conjunction:

$$\frac{\begin{array}{c} B\ b_1\ s_1\ t_1\ s_2 \\ B\ b_2\ s_2\ t_2\ s_3 \end{array}}{B\ (b_1 \wedge b_2)\ s_1\ (t_1 \wedge t_2)\ s_3}$$

Disjunction:

$$\frac{\begin{array}{c} B\ b_1\ s_1\ t_1\ s_2 \\ B\ b_2\ s_2\ t_2\ s_3 \end{array}}{B\ (b_1 \vee b_2)\ s_1\ (t_1 \vee t_2)\ s_3}$$

Negation:

$$\frac{B\ b\ s_1\ t\ s_2}{B\ (\sim b)\ s_1\ (\sim t)\ s_2}$$

Table 5.10: Boolean Expression Structural Operational Semantics.

In Table 5.11, we present rules that inductively define the command semantic relation C. This is a structural operational semantics for commands.

Skip:

$$\frac{}{C\ \textbf{skip}\ \rho\ s\ s}$$

Abort:

(no rules)

Assignment:

$$\frac{E\ e\ s_1\ n\ s_2}{C\ (x := e)\ \rho\ s_1\ s_2[n/x]}$$

Sequence:

$$\frac{C\ c_1\ \rho\ s_1\ s_2, \qquad C\ c_2\ \rho\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ \rho\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c_1\ \rho\ s_2\ s_3}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ s_3}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2, \qquad C\ c_2\ \rho\ s_2\ s_3}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ s_3}$$

Iteration:

$$\frac{\begin{array}{l} B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c\ \rho\ s_2\ s_3 \\ C\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_3\ s_4 \end{array}}{\begin{array}{l} C\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ s_4 \end{array}}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2}{\begin{array}{l} C\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ s_2 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \\ C\ (c \lhd [xs\ \&\ vals'/vars\ \&\ vals])\ \rho\ s_2[ns/vals']\ s_3 \end{array}}{C\ (\textbf{call}\ p(xs;\ es))\ \rho\ s_1\ s_3[(\textbf{map}\ s_2\ vals')/vals']}$$

Table 5.11: Command Structural Operational Semantics.

In Table 5.12, we present rules that inductively define the declaration semantic relation D. This is a structural operational semantics for declarations.

Procedure Declaration:

$$\frac{}{\begin{array}{r} D\ (\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c)\ \rho \\ \rho[\langle vars, vals, glbs, pre, post, calls, rec, c\rangle/p] \end{array}}$$

Declaration Sequence:

$$\frac{D\ d_1\ \rho_1\ \rho_2, \qquad D\ d_2\ \rho_2\ \rho_3}{D\ (d_1\ ;\ d_2)\ \rho_1\ \rho_3}$$

Empty Declaration:

$$\frac{}{D\ (\mathbf{empty})\ \rho\ \rho}$$

Table 5.12: Declaration Structural Operational Semantics.

In Table 5.13, we present rules that inductively define the program semantic relation P. This is a structural operational semantics for programs. As used in this definition, we define ρ_0 as the empty environment

$$\rho_0 = \lambda p.\ \langle [\,],\ [\,],\ [\,],\ \mathbf{false},\ \mathbf{true},\ (\lambda p.\ \mathbf{false}),\ \mathbf{false},\ \mathbf{abort}\rangle,$$

and s_0 as the initial state $s_0 = \lambda x.\ 0$.

Program:

$$\frac{D\ d\ \rho_0\ \rho_1, \qquad C\ c\ \rho_1\ s_0\ s_1}{P\ (\mathbf{program}\ d;\ c\ \mathbf{end\ program})\ s_1}$$

Table 5.13: Program Structural Operational Semantics.

5.5.2 Assertion Language Denotational Semantics

The denotational semantics of the three kinds of Sunrise assertion language phrases is expressed by the three functions in Table 5.14.

$V\ v\ s$	numeric expression v:`vexp` evaluated in state s
	yields numeric value in `num`
$VS\ vs\ s$	list of numeric expressions vs:`(vexp)list` evaluated in state s
	yields list of numeric values in `(num)list`
$A\ a\ s$	boolean expression a:`aexp` evaluated in state s
	yields truth value in `bool`

Table 5.14: Sunrise assertion language semantic functions.

In Table 5.15, we present a denotational definition of the assertion language semantic function V for numeric expressions.

$$\begin{array}{lcl}
V\ n\ s & = & n \\
V\ x\ s & = & s\ x \\
V\ (v_1 + v_2)\ s & = & V\ v_1\ s + V\ v_2\ s \\
V\ (v_1 - v_2)\ s & = & V\ v_1\ s - V\ v_2\ s \\
V\ (v_1 * v_2)\ s & = & V\ v_1\ s * V\ v_2\ s
\end{array}$$

Table 5.15: Assertion Numeric Expression Denotational Semantics.

In Table 5.16, we present a denotational definition of the assertion language semantic function VS for lists of numeric expressions.

$$
\begin{array}{lcl}
VS\ \langle\rangle\ s & = & [\,] \\
VS\ (CONS\ v\ vs)\ s & = & CONS\ (V\ v\ s)\ (VS\ vs\ s)
\end{array}
$$

Table 5.16: Assertion Numeric Expression List Denotational Semantics.

In Table 5.17, we present a denotational definition of the assertion language semantic function A for boolean expressions.

$$
\begin{array}{lcl}
A\ \mathbf{true}\ s & = & \mathrm{T} \\
A\ \mathbf{false}\ s & = & \mathrm{F} \\
A\ (v_1 = v_2)\ s & = & (V\ v_1\ s = V\ v_2\ s) \\
A\ (v_1 < v_2)\ s & = & (V\ v_1\ s < V\ v_2\ s) \\
A\ (vs_1 \ll vs_2)\ s & = & (VS\ vs_1\ s \ll VS\ vs_2\ s) \\
A\ (a_1 \wedge a_2)\ s & = & (A\ a_1\ s \wedge A\ a_2\ s) \\
A\ (a_1 \vee a_2)\ s & = & (A\ a_1\ s \vee A\ a_2\ s) \\
A\ (\sim a)\ s & = & \sim(A\ a\ s) \\
A\ (a_1 \Rightarrow a_2)\ s & = & (A\ a_1\ s \Rightarrow A\ a_2\ s) \\
A\ (a_1 = a_2)\ s & = & (A\ a_1\ s = A\ a_2\ s) \\
A\ (a_1 => a_2 \mid a_3)\ s & = & (A\ a_1\ s => A\ a_2\ s \mid A\ a_3\ s) \\
A\ (\mathbf{close}\ a)\ s & = & (\forall s_1.\ A\ a\ s_1) \\
A\ (\forall x.\ a)\ s & = & (\forall n.\ A\ a\ s[n/x]) \\
A\ (\exists x.\ a)\ s & = & (\exists n.\ A\ a\ s[n/x])
\end{array}
$$

Table 5.17: Assertion Boolean Expression Denotational Semantics.

The lexicographic ordering $\ll$ is defined as

$$
\begin{array}{rcl}
[\,] \ll [\,] & = & \mathrm{F} \\
[\,] \ll CONS\ n\ ns & = & \mathrm{T} \\
CONS\ n\ ns \ll [\,] & = & \mathrm{F} \\
CONS\ n_1\ ns_1 \ll CONS\ n_2\ ns_2 & = & n_1 < n_2\ \vee\ (n_1 = n_2\ \wedge\ ns_1 \ll ns_2)
\end{array}
$$

This concludes the definition of the semantics of the assertion language.

The Sunrise language is properly thought of as consisting of both the programming language *and* the assertion language, even though the assertion language is never executed, and only exists to express specifications and annotations, to facilitate proofs of correctness. The two languages are different in character; the semantics of the programming language is very dependent on time; it both responds to and causes the constantly changing state of the memory. In contrast, the assertion language has a timeless quality, where, for a given state, an expression will always evaluate to the same value irrespective of how many times it is evaluated. The variables involved also reflect this, where program variables often change their values during execution, but logical variables never do. The programming language is an active, involved participant in the execution as it progresses; the assertion language takes the role of a passive, detached observer of the process.

This difference carries over to how the languages are used. States and their changes in time are the central focus of the operational semantics, whereas assertions and their permanent logical interrelationships are the focus of the axiomatic semantics. Programs in the programming language are executed, causing changes to the state. Assertions in the assertion language are never executed or even evaluated. Instead they are stepping stones supporting the proofs of correctness, which also have a timeless quality. Done once for all possible executions of the program, a proof replaces and exceeds any finite number of tests.

5.6 Procedure Entrance Semantic Relations

In addition to the traditional structural operational semantics of the Sunrise programming language, we also define two semantic relations that connect to states reached at the entrances of procedures called from within a command. These semantic relations are used to define the correctness specifications for the Entrance Logic.

The entrance structural operational semantics of commands and procedures is expressed by the two relations described in Table 5.18.

$C\_calls\ c\ \rho\ s_1\ p\ s_2$	Command c:`cmd`, evaluated in environment ρ and state s_1, calls procedure p directly from c, where the state just after entering p is s_2.
$M\_calls\ p_1\ s_1\ ps\ p_2\ s_2\ \rho$	The body of procedure p_1, evaluated in environment ρ and state s_1, goes through a path ps of successively nested calls, and finally calls p_2, where the state just after entering p_2 is s_2.

Table 5.18: Sunrise programming language entrance semantic relations.

In Table 5.19, we present rules that inductively define the command semantic relation C_calls.

Skip:

(no rules)

Abort:

(no rules)

Assignment:

(no rules)

Sequence:

$$\frac{C\_calls\ c_1\ \rho\ s_1\ p\ s_2}{C\_calls\ (c_1\ ;\ c_2)\ \rho\ s_1\ p\ s_2}$$

$$\frac{\begin{array}{c} C\ c_1\ \rho\ s_1\ s_2 \\ C\_calls\ c_2\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (c_1\ ;\ c_2)\ \rho\ s_1\ p\ s_3}$$

Conditional:

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C\_calls\ c_1\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ p\ s_3}$$

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{F}\ s_2 \\ C\_calls\ c_2\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ p\ s_3}$$

Iteration:

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C\_calls\ c\ \rho\ s_2\ p\ s_3 \end{array}}{\begin{array}{l} C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ p\ s_3 \end{array}}$$

$$\frac{\begin{array}{l} B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c\ \rho\ s_2\ s_3 \\ C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_3\ p\ s_4 \end{array}}{\begin{array}{l} C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ p\ s_4 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \end{array}}{C\_calls\ (\textbf{call}\ p\ (xs; es))\ \rho\ s_1\ p((s_2[ns/vals']) \lhd [xs\ \&\ vals'/vars\ \&\ vals])}$$

Table 5.19: Command Entrance Semantic Relation.

In Table 5.20, we present rules that inductively define the procedure path semantic relation M_calls.

Single:

$$\frac{\begin{array}{c}\rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ C\_calls\ c\ \rho\ s_1\ p_2\ s_2\end{array}}{M\_calls\ p_1\ s_1\ []\ p_2\ s_2\ \rho}$$

Multiple:

$$\frac{\begin{array}{c}M\_calls\ p_1\ s_1\ ps_1\ p_2\ s_2\ \rho \\ M\_calls\ p_2\ s_2\ ps_2\ p_3\ s_3\ \rho\end{array}}{M\_calls\ p_1\ s_1\ (ps_1\ \&\ (CONS\ p_2\ ps_2))\ p_3\ s_3\ \rho}$$

Table 5.20: Path Entrance Semantic Relation.

5.7 Termination Semantic Relations

In addition to the other structural operational semantics of the Sunrise programming language, we also define two semantic relations that describe the termination of executions begun in states reached at the entrances of procedures called from within a command. These semantic relations are used to define the correctness specifications for the Termination Logic.

The termination semantics of commands and procedures is expressed by the two relations in Table 5.21. These termination semantic relations are true when all direct calls from c or from the body of p_1 are known to terminate.

$C\_calls\_terminate\ c\ \rho\ s_1$	For every procedure p and state s_2 such that
	$C\_calls\ c\ \rho\ s_1\ p\ s_2$,
	the body of p executed in state s_2 terminates.
$M\_calls\_terminate\ p_1\ s_1\ \rho$	For every procedure p_2 and state s_2 such that
	$M\_calls\ p_1\ s_1\ [\,]\ p_2\ s_2\ \rho$,
	the body of p_2 executed in state s_2 terminates.

Table 5.21: Sunrise programming language termination semantic relations.

In Tables 5.22 and 5.23, we present the definitions of the command termination semantic relation *C_calls_terminate* and the procedure path termination semantic relation *M_calls_terminate*.

$$\begin{array}{l}
C\_calls\_terminate\ c\ \rho\ s_1\ \ = \\
\qquad \forall p\ s_2.\ C\_calls\ c\ \rho\ s_1\ p\ s_2\ \Rightarrow \\
\qquad\qquad (\exists s_3.\ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c' \rangle = \rho\ p\ \mathbf{in} \\
\qquad\qquad\quad C\ c'\ \rho\ s_2\ s_3)
\end{array}$$

Table 5.22: Command Termination Semantic Relation *C_calls_terminate*.

$$\begin{array}{l}
M\_calls\_terminate\ p_1\ s_1\ \rho\ \ = \\
\qquad \forall p_2\ s_2.\ M\_calls\ p_1\ s_1\ [\,]\ p_2\ s_2\ \rho\ \Rightarrow \\
\qquad\qquad (\exists s_3.\ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c \rangle = \rho\ p_2\ \mathbf{in} \\
\qquad\qquad\quad C\ c\ \rho\ s_2\ s_3)
\end{array}$$

Table 5.23: Procedure Path Termination Semantic Relation *M_calls_terminate*.

The definitions of the relations presented in this chapter define the semantics of the Sunrise programming language, as a foundation for all later work. From this point on, all descriptions of the meanings of program phrases will be proven as theorems from this foundation, with the proofs mechanically checked. This will ensure the soundness of the later axiomatic semantics, a necessary precondition to a verified VCG.